F#

Succinctly

by Robert Pickering

Syncfusion | Technology Resource Portal

Chapter 1 Introduction

This introductory chapter will address some of the major questions you may have about F# and functional programming (FP).

What Is Functional Programming?

Pure functional programming views all programs as collections of functions that accept arguments and return values. Unlike imperative and object-oriented programming, it allows no side effects and uses recursion instead of loops for iteration. The functions in a functional program are very much like mathematical functions because they do not change the state of the program. In the simplest terms, once a value is assigned to an identifier it never changes, functions do not alter parameter values, and the results that functions return are completely new values. In typical underlying implementations, once a value is assigned to an area in memory, it does not change. To create results, functions copy values and then change the copies, leaving the original values free to be used by other functions and eventually be thrown away when no longer needed. (This is where the idea of garbage collection originated.)

The mathematical basis for pure functional programming is elegant, and FP therefore provides beautiful, succinct solutions for many computing problems, but its stateless and recursive nature makes the other paradigms convenient for handling many common programming tasks. However, one of F#'s great strengths is that you can use multiple paradigms and mix them to solve problems in the way you find most convenient.

Why Is Functional Programming Important?

When people think of functional programming, they often view its statelessness as a fatal flaw without considering its advantages. One could argue that since an imperative program is often 90 percent assignment, and a functional program has no assignment, a functional program could be 90 percent shorter. However, not many people are convinced by such arguments or attracted to the ascetic world of stateless recursive programming, as John Hughes pointed out in his classic paper "Why Functional Programming Matters."

> *The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous.*
>
> John Hughes, Chalmers University of Technology
> (http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html)

To see the advantages of functional programming, you must look at what FP permits rather than what it prohibits. For example, functional programming allows you to treat functions themselves as values and pass them to other functions. This might not seem all that important at first glance, but its implications are extraordinary. Eliminating the distinction between data and function means that many problems can be more naturally solved. Functional programs can be shorter and more modular than corresponding imperative and object-oriented programs.

In addition to treating functions as values, functional languages offer other features that borrow from mathematics and are not commonly found in imperative languages. For example, functional programming languages often offer *curried functions*, where arguments can be passed to a function one at a time and, if all arguments are not given, the result is a residual function waiting for the rest of its parameters. It's also common for functional languages to offer type systems with much better power-to-weight ratios, providing more performance and correctness for less effort.

What Is F#?

Functional programming is the best approach to solving many thorny computing problems, but pure FP often isn't suitable for general-purpose programming. Because of this, FP languages have gradually embraced aspects of the imperative and OO paradigms, remaining true to the FP paradigm but incorporating features needed to easily write any kind of program. F# is a natural successor on this path. It is also much more than just an FP language.

Some of the most popular functional languages, including OCaml, Haskell, Lisp, and Scheme, have traditionally been implemented using custom runtimes, which leads to problems such as lack of interoperability. F# is a general-purpose programming language for .NET, a general-purpose runtime. F# smoothly integrates all three major programming paradigms. With F#, you can choose whichever paradigm works best to solve problems in the most effective way. You can do pure functional programming if you're a purist, but you can easily combine functional, imperative, and object-oriented styles in the same program and exploit the strengths of each paradigm. Like other typed functional languages, F# is strongly typed but also uses inferred typing so programmers don't need to spend time explicitly specifying types unless an ambiguity exists. Further, F# seamlessly integrates with the .NET Framework base class library (BCL). Using the BCL in F# is as simple as using it in C# or Visual Basic (and maybe even simpler).

F# was modeled on Objective Caml (OCaml), a successful object-oriented functional programming language, and then tweaked and extended to mesh well technically and philosophically with .NET. It fully embraces .NET and enables users to do everything that .NET allows. The F# compiler can compile for all implementations of the Common Language Infrastructure (CLI), it supports .NET generics without changing any code, and it even provides for inline Intermediate Language (IL) code. The F# compiler not only produces executables for any CLI, but can also run on any environment that has a CLI, which means F# is not limited to Windows but can run on Linux, Apple's OS X and iOS, as well as Google's Android OS.

The F# 2.0 compiler is distributed with Visual Studio 2012, Visual Studio 2010, and available as a plug-in for Visual Studio 2008. It supports IntelliSense expression completion and automatic expression checking. It also gives tooltips to show what types have been inferred for expressions. Programmers often comment that this really helps bring the language to life. F# 2.0 also has an open source release, licensed under the Apache License and is available from http://github.com/fsharp.

F# was fist implemented by Don Syme at Microsoft Research (MSR) in Cambridge. The project has now been embraced by Microsoft Corporate in Redmond, WA and the implementation of the compiler and Visual Studio integration is now developed by a team located in both Cambridge and Redmond. At the time of writing, the team was focused implementing F# 3.0, which is available in the Visual Studio "dev11" beta.

Although other FP languages run on .NET, F# has established itself as the de facto .NET functional programming language because of the quality of its implementation and its superb integration with .NET and Visual Studio.

No other .NET language is as easy to use and as flexible as F#!

Who Is Using F#?

F# has a strong presence inside Microsoft, both in MSR and throughout the company as a whole. Ralf Herbrich, coleader of MSR's Applied Games Group, which specializes in machine learning techniques, is typical of F#'s growing number of fans:

> *The first application was parsing 110GB of log data spread over 11,000 text files in over 300 directories and importing it into a SQL database. The whole application is 90 lines long (including comments!) and finished the task of parsing the source files and importing the data in under 18 hours; that works out to a staggering 10,000 log lines processed per second! Note that I have not optimized the code at all but written the application in the most obvious way. I was truly astonished as I had planned at least a week of work for both coding and running the application.*
>
> *The second application was an analysis of millions of feedbacks. We had developed the model equations and I literally just typed them in as an F# program; together with the reading-data-from-SQL-database and writing-results-to-MATLAB-data-file, the F# source code is 100 lines long (including comments). Again, I was astonished by the running time; the whole processing of the millions of data items takes 10 minutes on a standard desktop machine. My C# reference application (from some earlier tasks) is almost 1,000 lines long and is no faster. The whole job from developing the model equations to having first real world data results took 2 days.*
>
> Ralf Herbrich, Microsoft Research
> (http://blogs.msdn.com/dsyme/archive/2006/04/01/566301.aspx)

F# usage outside Microsoft is also rapidly growing. I asked Chance Coble, CTO at Cyfeon Solutions, about what F# brought to his work.

> *F# has made its case to me over and over again. The first project I decided to try F# on was a machine vision endeavor, which would identify and extract fingerprints from submitted fingerprint cards and load them into a biometrics system. The project plan was to perform the fingerprint extraction manually, which was growing cumbersome and the automation turned out to be a huge win (with very little code). Later we decided to include that F# work in a larger application that had been written in C#, and accomplished the integration with ease. Since then I have used F# in projects for machine learning, domain-specific language design, 3-D visualizations, symbolic analysis, and anywhere performance intensive data processing has been required. The ability to easily integrate functional modules into existing production scale applications makes F# not only fun to work with, but an important addition for project leads. Unifying functional programming with a mature and rich platform like .NET has opened up a great deal of opportunity.*
>
> Chance Coble, CTO at Cyfeon Solutions (private email)

Chapter 1 Introduction

This introductory chapter will address some of the major questions you may have about F# and functional programming (FP).

What Is Functional Programming?

Pure functional programming views all programs as collections of functions that accept arguments and return values. Unlike imperative and object-oriented programming, it allows no side effects and uses recursion instead of loops for iteration. The functions in a functional program are very much like mathematical functions because they do not change the state of the program. In the simplest terms, once a value is assigned to an identifier it never changes, functions do not alter parameter values, and the results that functions return are completely new values. In typical underlying implementations, once a value is assigned to an area in memory, it does not change. To create results, functions copy values and then change the copies, leaving the original values free to be used by other functions and eventually be thrown away when no longer needed. (This is where the idea of garbage collection originated.)

The mathematical basis for pure functional programming is elegant, and FP therefore provides beautiful, succinct solutions for many computing problems, but its stateless and recursive nature makes the other paradigms convenient for handling many common programming tasks. However, one of F#'s great strengths is that you can use multiple paradigms and mix them to solve problems in the way you find most convenient.

Why Is Functional Programming Important?

When people think of functional programming, they often view its statelessness as a fatal flaw without considering its advantages. One could argue that since an imperative program is often 90 percent assignment, and a functional program has no assignment, a functional program could be 90 percent shorter. However, not many people are convinced by such arguments or attracted to the ascetic world of stateless recursive programming, as John Hughes pointed out in his classic paper "Why Functional Programming Matters."

> *The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous.*
>
> John Hughes, Chalmers University of Technology
> (http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html)

To see the advantages of functional programming, you must look at what FP permits rather than what it prohibits. For example, functional programming allows you to treat functions themselves as values and pass them to other functions. This might not seem all that important at first glance, but its implications are extraordinary. Eliminating the distinction between data and function means that many problems can be more naturally solved. Functional programs can be shorter and more modular than corresponding imperative and object-oriented programs.

Who Is This Book For?

This book is aimed primarily at IT professionals who want to get up to speed quickly on F#. A working knowledge of the .NET Framework and some knowledge of either C# or Visual Basic would be nice, but it's not necessary. All you really need is some experience programming in any language to be comfortable learning F#.

Even complete beginners who've never programmed before and are learning F# as their first computer language should find this book very readable. Though it doesn't attempt to teach introductory programming per se, it does carefully present all the important details of F#.